• **Iteration/repetition/looping**: Executing a sequence of instructions multiple times

• Loop components:
  – Condition: determines whether loop should continue
  – Body: instructions being repeated
  – Loop control variable (LCV): Variable whose value determines when loop stops

• Basic steps of loop control:

  Initialize Process
  Initialize LCV

  BEGIN LOOP
  Update Process
  Update LCV
  END LOOP

• Two types of loops:

  1. Pretest
     – Condition precedes body
     – Iterates 0+ times
2. Posttest
   – Condition follows body
   – Iterates 1+ times
Python: Iteration - While Loops

- Pretest loop
- Most general loop construct
  - Only loop construct needed
- Syntax:
  ```python
  while <condition>:
    <body>
  [else:
    <post_code>]
  ```
  - body and post_code are blocks
- Semantics:

  ![Loop Diagram]

  - The else/post_code is rarely used as the above is equivalent to

    ```python
    while <condition>:
      <body>
      [<post_code>]
    ```
Python: Iteration - Loop Control

• Main issue with loops is *loop control*
  
  – Determining
    1. How many times the loop body should execute, and
    2. The methods for controlling this

• 2 general types of loop control:
  1. Definite iteration -
     – Loop executes a fixed number of times
  2. Indefinite iteration -
     – Loop executes an indeterminate number of times
Python: Iteration - Definite Iteration

• Is a variation of count-controlled iteration

• Body will execute a fixed number of times

• LCV is an example of a variable called a counter
  – It counts how many times something occurs

• Example

```
# count-controlled while loop: Average <limit> integers

limit = 10
counter = 0
sum = 0
while counter < limit:
    number = int(input('Enter an integer: '))
    counter = counter + 1
    sum += number
print("The average = ", sum / limit)
```

• counter is the LCV

• counter is a counter variable: it counts how many times the loop body has executed

• Sum is an accumulator: it holds a value that is augmented on each iteration
Python: Iteration - Indefinite Iteration

- Used when don’t know ahead of time how many times loop should execute

1. Count controlled
   - Iterations determined by occurrence of some event a specified number of times
   - Loop may need to iterate more than this number
   - Example:
     ```python
     # count-controlled while loop, indefinite iteration -
     # Average <limit> positive integers

     limit = 10
     counter = 0
     sum = 0
     while counter < limit:
         number = int(input('Enter an integer: '))
         if number >= 0:
             counter = counter + 1
             sum += number
     print("The average = ", sum / limit)
     ```

2. Sentinel control
   - *Sentinel* is one or more values that signal the end of iteration
   - Usually outside of range of acceptable values used for processing
   - Example:
     ```python
     # Sentinel-controlled while loop: Average non-negative
     # integers. Terminate when a negative is input.

     counter = 0
     sum = 0
     number = int(input('Enter an integer: '))
     while number >= 0:
         counter = counter + 1
         sum += number
         number = int(input('Enter an integer: '))
     if counter == 0:
         print("no integers were averaged")
     else:
         print("The average = ", sum / counter)
     ```
   - Requires *priming read*
     * Must get initial value for *LCV* prior to entering loop
     * Note that loop body may never execute if priming value is negative
3. Flag control

- *Flag* is Boolean value used as an *LCV*
- It keeps track of whether some situation ”of interest” has occurred
- Can be thought of as a specialized sentinel
- Often used when multiple situations are used to terminate loop

Example:

```python
# Flag-controlled while loop: Determine if an int is prime.
# Terminate when divisor = number or if is prime

x = eval(input("Enter an integer: "))
done = False
div = 2
is_prime = True
while (not done):
    y = x % div
    if y == 0:
        is_prime = False
        done = True
    div += 1
    if div == x:
        done = True
if is_prime:
    print(x, " is a prime number")
else:
    print(x, " is not a prime number")
```
Python: Iteration - Potential Problems

1. Infinite loop
   
   • Loop that never ends
   
   • Can result due to many reasons:
     – Failure to update $LCV$
     – Failure to update $LCV$ properly
     – Poor choice of termination condition
     – Failure to update process

2. Off-by-one error

   • Count-controlled loop that iterates one too many or one too few times
Python: Iteration - Nested Loops

• Can place one loop inside of another

• Must make sure that initializations and updates occur for each loop and in the appropriate places

• Example:

```python
# nested WHILE loops: Print times tables.

n = 5
counter1 = 1
while counter1 <= n:
    print(counter1, "'s times table: ")
counter2 = 1
    while counter2 <= 9:
        print(counter1, " x ", counter2, " = ", end = "")
        print(counter1 * counter2)
        counter2 = counter2 + 1
    counter1 = counter1 + 1
```
Python: Iteration - *Break* and *Continue*

- In some situations, it makes sense to halt loop execution in middle of loop
- The *break* statement immediately exits the body of a loop
  - Control passes to the statement that follows the loop (remember that the optional *else/post_code* is part of the loop, so it is bypassed also)
Python: Iteration - *Break* and *Continue* (2)

- *break* is usually part of an *if* statement
- For example (using earlier *flag* example):

```python
# Determine if an int is prime.
# Terminate when divisor = number or if is prime

x = eval(input("Enter an integer: "))
div = 2
is_prime = True
while (div < x):
y = x % div
if y == 0:
    is_prime = False
    break
div += 1
if is_prime:
    print(x, " is a prime number")
else:
    print(x, " is not a prime number")
```

- Problems:
  1. Potential infinite loop
  2. More difficult to debug as now have multiple exit points
Python: Iteration - *Break* and *Continue* (3)

- The *continue* statement immediately jumps to the end of the body of a loop
  - The loop continues from this point

```python
# count-controlled while loop, indefinite iteration -
# Average <limit> positive integers

limit = 10
counter = 0
sum = 0
while counter < limit:
    number = int(input('Enter an integer: '))
    if number < 0:
        continue
    counter = counter + 1
    sum += number
print("The average = ", sum / limit)
```
Python: Iteration - Error-checking Input

• Many programs rely on user input

• This can be a problem if the user types in the wrong kind of value:
  – A value that’s not expected, like a negative value for a grade
  – A value of the wrong data type, like a string instead of a number

• You can always check if a value is incorrect using an *if* statement:

  ```python
  x = eval(input('Enter a number between 0 and 10: '))
  if x < 0 or x > 10:
      x = eval(input('Incorrect; Enter a number between 0 and 10: '))
  ```

  – But what if they do it again?

• A better way is to use a loop

  ```python
  x = eval(input('Enter a number between 0 and 10: '))
  while x < 0 or x > 10:
      x = eval(input('Incorrect; Enter a number between 0 and 10: '))
  ```

• Usually a wrong data type is more of a problem than a value of the correct type but with the wrong value
  – With a wrong value the program will most likely still run

• To test data types use the *type* function, which returns the type of its argument

  ```python
  x = eval(input('Please enter an integer: '))
  while type(x) != int:
      print(x,'is not an integer: Try again: ', end = '')
      x = eval(input())
  ```

• You can also use *is* and *is not* in place of *==* and *!=*
Python: Iteration - *For* Loops

- **Pre test loop**
- **Used to iterate over elements in a sequence (string, list, tuple, ...)**
- **Essentially count-controlled looping** - iterates once for each element in the sequence
- **Syntax:**

  ```python
  for <lcv> in <sequence>:
      <body>
  else:
      <post_code>
  ```

  - As with *while* loops, the *post_code* is rarely included
- **Semantics:**

  ```python
  lcv = next element in sequence
  while lcv != end_of_list:  # NOT last element of list
      body
      lcv = next element in sequence
  ```

  - Think of *lcv* starting *in front* of the first element in the list
  - Think of *end_of_list* as *after* of the last element in the list
- **Example:**

  ```python
  # Average a list of integers.
  lst = [8, 1, 3, 6]
  limit = len(lst)
  sum = 0
  for n in lst:
      sum += n
  print("The average = ", sum / limit)
  ```

- **Packing and unpacking**

  - If an iterable’s elements are themselves iterables, a tuple can be used for the iterator variable to access all components of each element in the *for* loop:

  ```python
  lst = [[1, 2, 3], [4, 5, 6]]
  for m, n, o in lst:
      print(m + n + o)
  ```

  will print

  6
  15
Python: Iteration - The Range Function

• The range function generates a series of integers
• Syntax: range(<e1>[, <e2>][, <e3>])
• Semantics
  – Depends on how many parameters are present
    1. range(<e1>)
       * Generates a series of e1 integers from 0 to e1 − 1
       * Example:
         >>> x = list(range(5))
         >>> x
         >>> [0, 1, 2, 3, 4]
    2. range(<e1>, <e2>)
       * Generates a series of integers from e1 to e2 − 1
       * Example:
         >>> x = list(range(5, 10))
         >>> x
         >>> [5, 6, 7, 8, 9]
       * If e1 ≥ e2 nothing is generated
         >>> x = list(range(5, 5))
         >>> x
         >>> []
    3. range(<e1>, <e2>, <e3>)
       * Generates a series of integers from e1 to e2 − 1, changing each successive value by e3
       * Example:
         >>> x = list(range(5, 10, 2))
         >>> x
         >>> [5, 7, 9]
       * This allows you to generate numbers in decreasing order
         >>> x = list(range(10, 5, -1))
         >>> x
         >>> [10, 9, 8, 7, 6]
• *Range* can be used with *for* as an alternative to using *while* to visit each element of a list when you need to know the index of the element you’re visiting
  
  – For example
  ```python
  for n in range(len(x)):
      #process x[n]
  ```

  – A practical example:
  ```python
  # Make a list of positions of integers and floats in a list
  ans = []
  for n in range(len(x)):
      if (type(x[n]) is int) or (type(x[n]) is float):
          ans.append(n)
  ```

  – If *x* = [‘*hello*’, 3, 10.2, (3 + 6j), 1, 6,’*goodbye’*], then *ans* will have the value [1, 2, 4, 5]
Python: Iteration - The *Enumerate* and *Zip* Functions

- **Enumerate**
  - Syntax: `enumerate(< iterable >)`
  - Semantics: Creates a list of tuples consisting of each element of *iterable* and its index (index first)
  - For example:
    ```python
    >>> x = [3, 6, 2, 7, 9]
    >>> list(enumerate(x))
    >>> [(0, 3), (1, 6), (2, 2), (3, 7), (4, 9)]
    ```
    - This technique could have been used in the earlier `range` example
      ```python
      # Make a list of positions of integers and floats in a list
      ans = []
      for m, n in enumerate(x):
        if (type(n) is int) or (type(n) is float):
          ans.append(m)
      ```

- **Zip**
  - Syntax: `zip(< iterable > ...)`
  - Semantics: Creates a list of tuples consisting of successive elements from each of the *iterable* arguments
    There will be as many tuples as the *shortest* iterable argument
  - For example:
    ```python
    >>> x = [3, 6, 2, 7, 9]
    >>> y = ['a', 'b', 'c']
    >>> z = [1.1, 2.2, 3.3, 4.4]
    >>> list(zip(x, y, z))
    >>> [(3, 'a', 1.1), (6, 'b', 2.2), (2, 'c', 3.3)]
    ```
Python: Iteration - The *Enumerate* and *Zip* Functions (2)

This technique is useful if you need to go through lists in tandem:

# Find largest integer at each position of multiple lists

```python
x = [3, 6, 2, 7, 9]
y = [4, 1, 7, 8, 3]
z = [1, 5, 9, 2, 6]
ans = []
for a, b, c in zip(x, y, z):
    if a >= b >= c:
        ans.append(a)
    elif b >= c >= a:
        ans.append(b)
    else:
        ans.append(c)
```

will generate

```
[4, 6, 9, 8, 9]
```